En Français

JAVASCRIPT ES6

de A à Z

Julien CROUZET

Table des matières

Javascript ES6 de A à Z

(Re) Découvrez la version 6 de ECMAScript, ou ES6, la nouvelle version de Javascript, en détail. (et en Français)

La déclaration de variable

Jusqu'à EcmaScript 5, la déclaration des variables en Javascript se fesait par l'intermédiaire du instruction `var`.

```
var {nom de la variable} [= valeur][, var2 [= valeur2]] [, ...];
// Ex :
var resultat;
var nombreDeTours = 42;
var hello = "world", foo = { bar: 1 };
```

Cette instruction déclare une ou plusieurs variable nommée (`{nom de la variable}`) pour l'ensemble de la fonction qui la contient ; avec ou sans valeur initiale (`= valeur`).

Portée de la variable

Une variable déclarée à l'aide de `var` a une portée sur l'ensemble de la fonction ou elle est déclarée :

```
function maFonction() {
  var maVariable = 42;

  function sousFonction() {
    console.log(maVariable);
    // => 42
  }
  console.log(maVariable);
  // => 42
  sousFonction();
}

maFonction();
console.log(maVariable);
// => ReferenceError: maVariable is not defined
```

Lorsqu'une variable est déclarée à l'aide de `var` en dehors d'une fonction, elle crée un propriété à l'objet global (`window` dans un navigateur, `global` ou `module` dans Node.js, etc).

```
// Dans un navigateur

var maVariable = 42;

console.log(window.maVariable);
// => 42
```

Le principe du "hoisting"

Le "hoisting" (ou "hissage" en français) est une allégorie décrivant la façon dont Javascript gère les variables au sein de la fonction ou elles sont déclarées ; en effet, peut importe la ligne à laquelle se trouve l'instruction `var`, la variable est déclarée "jusqu'en haut" de la fonction.

Par exemple, si l'on considère l'exemple suivant :

```
(function() {
  console.log(variableNonDeclaree);
})();
// => ReferenceError: variableNonDeclaree is not defined
```

Si l'instruction `var variableNonDeclaree` est présente, même après l'instruction `console.log()`, la variable sera déclarée pour toute la fonction.

```
(function() {
  console.log(variableNonDeclaree);
  var variableNonDeclaree;
})();
// => undefined
```

A noter cependant si la valeur initiale est présente dans l'instruction `var`, la variable sera définie sur l'ensemble de la fonction, mais n'aura de valeur qu'après sa déclaration :

```
(function() {
  console.log(variableNonDeclaree);
  var variableNonDeclaree = 42;
  console.log(variableNonDeclaree);
})();
// => undefined
// => 42
```

Le concept de "hoisting" couplé à la portée des variables n'etant pas naturels à la lecture naturelle du code, cela peut introduire des bugs de conception, notamment en présence de code asynchrone.

Prenons l'exemple suivant qui va lancer le téléchargement de trois fichiers et afficher leur contenu :

```
// Fonction simulant le téléchargement d'un fichier
function telecharge(url, callback) {
  setTimeout(function() { callback('contenu') }, 500);
}

function telechargeTous() {
  for (var numero = 1; numero <= 3; numero++) {
    telecharge('http://url.com/' + numero, function(contenu) {
      console.log('Contenu du fichier ' + numero + ' => ' + contenu);
    })
  }
}
telechargeTous();
```

Le résultat affiché sera le suivant :

```
"Contenu du fichier 3 => contenu"
"Contenu du fichier 3 => contenu"
"Contenu du fichier 3 => contenu"
```

Comme vous pouvez le constater, la variable `numero` vaut `3` dans les trois messages ; en effet, elle est déclarée pour l'ensemble de la fonction `telechargeTous()` et non uniquement la boucle `for` .

Le callback de la fonction `telecharge()` n'étant éxécuté qu'après la fin de la boucle `for` il est donc resté à sa dernière valeur.

Redéclaration

Les variables peuvent être rédéclarées plusieurs fois avec l'instruction `var` au sein d'une même fonction ; tout en gardant la valeur précédente si la déclaration n'est pas accompagnée d'un valeur :

```
(function() {
    var variable = 42;

    for (var i = 0; i <= 10; i++) {
        var variable = i * 10;
    }
    console.log(variable);
    var variable;
    console.log(variable);
    var variable = 42;
})();
// => 100
// => 100
// => 42
```

A nouveau, cela peut introduire des bugs de conception en cas de fonction volumineuses, surtout si le développeur n'a pas le reflexe de déclarer toute les variables en haut de la fonction (En savoir plus).

La déclaration de variable `let`

Ecmascript 6 a introduit une nouvelle instruction de déclaration de variable : `let`, qui suit la même syntaxe que `var` :

```
let {nom de la variable} [= valeur][, var2 [= valeur2]] [, ...];
// Ex :
let resultat;
let nombreDeTours = 42;
let hello = "world", foo = { bar: 1 };
```

Portée limitée au bloc courant

A l'inverse de `var` qui déclare la variable pour l'ensemble de la fonction où elle est déclarée, `let` ne la déclare que pour le bloc courant.

Un bloc en Javascript peut être (un peu vulgairement) simplifié par *le code contenu entre deux accolades* : { `... code du bloc ...` }, par exemple le corps d'une fonction, ou le contenu d'un bloc `if`.

```
(function() {
  var variableVar = 42;
  let variableLet = 42;

  if (1) {
    var variableVar = 'bonjours'; // La même variable est redéclarée
    let variableLet = 'bonjours'; // C'est une nouvelle variable pour ce bloc

    console.log(variableVar);
    // => "bonjours"
    console.log(variableLet);
    // => "bonjours"
  }
  console.log(variableVar);
  // => "bonjours"
  console.log(variableLet);
  // => 42
})();
```

Dans le cadre d'une déclaration hors d'un bloc, c'est à dire dans le contexte global, `let` crée une vraie variable globale (là ou `var` créait une propriété de l'objet global) :

```
var variableVar = 42;
let variableLet = 42;

console.log(window.variableVar);
// => 42
console.log(window.variableLet);
// => undefined
```

Redéclaration

Les variables déclarées par `let` sont uniques dans un bloc courant et ne peuvent être redéclarées :

```
(function() {
  let variable = 42;
  // ... code ...
  if (1) {
    let variable = 'bonjours'; // Nouvelle variable pour ce bloc
    // ... code ...
  }
  let variable = 'ici ca marche';
  // => SyntaxError: Identifier 'variable' has already been declared
})();
```

Zone morte temporaire

Les variables déclarées avec `let` n'introduisent plus les effets du Hissage (hoisting) de `var`, étant placées dans une zone morte temporaire (souvent appelée *TDZ* ou *Temporary Dead Zone*) entre le début du bloc et la ligne ou elles sont déclarées.

Cette zone morte temporaire interdit toute référence au nom de la variable en provoquant une erreur de type `ReferenceError`.

```
(function() {
  // ... code ...
  console.log(maVariable);
  // => ReferenceError: maVariable is not defined
  // ... code ...
  let maVariable = 42;
})();
```

Utilisation de `let` dans les boucles `for`

La déclaration de variable avec `let` peut être utilisée dans l'initialisation d'une boucle `for` afin de limiter sa portée au bloc d'éxécution de la boucle :

```
for(let iteration = 0; iteration < 10; iteration++) {
  console.log(iteration);
  // => 0 ... 9
}
console.log(iteration);
// => ReferenceError: iteration is not defined
```

Cela permet par exemple d'éviter le bug décrit dans l'exemple de l'introduction du chapitre en remplaçant :

```
// Fonction simulant le téléchargement d'un fichier
function telecharge(url, callback) {
  setTimeout(function() { callback('contenu') }, 500);
}

function telechargeTous() {
  // Ici let remplace var
  for (let numero = 1; numero <= 3; numero++) {
    telecharge('http://url.com/' + numero, function(contenu) {
      console.log('Contenu du fichier ' + numero + ' => ' + contenu);
    })
  }
}
telechargeTous();
// "Contenu du fichier 1 => contenu"
// "Contenu du fichier 2 => contenu"
// "Contenu du fichier 3 => contenu"
```

La déclaration de variable `const`

En même temps que `let`, Ecmascript introduit également une nouvelle instruction de déclaration de variable : **`const`**.

Sa syntaxe diffère de `let` sur le fait que la valeur est obligatoire :

```
const {nom de la variable} = valeur[, var2 = valeur2] [, ...];
// Ex :
const nombreDeTours = 42;
const hello = "world", foo = { bar: 1 };
```

Comme on peut le deviner par son nom, les variables déclarées par `const` sont constantes ; cela signifie qu'une fois déclarée, cette variable n'est accessible qu'en lecture.

```
(function() {
  const maConstante = 42;

  maConstante = 12;
  // => TypeError: Assignment to constant variable.
})();
```

Sur certains moteurs javascript, il est possible que la ligne `maConstante = 12` ne génère pas d'erreur mais soit ignorée silencieusement ; dans ce cas là, `maConstante` reste égale à `42`.

`const` permet donc de s'assurer qu'une variable restera bien à la valeur assignée lors de sa déclaration et qu'elle ne sera pas modifiée.

A cette différence près, `const` obéit aux même règles que `let` (portée de bloc, redéclaration, zone morte temporaire).

Différence entre valeur et contenu

Le code suivante est tout à fait valable :

```
(function() {
  const monObjet = { foo: 'bar' };

  monObjet.hello = 'world';
  monObjet.foo = 42;
})();
```

L'instruction `const monObjet = { foo: 'bar' };` déclare une variable dont le nom est `monObjet` identifiant un objet `{ foo: 'bar' }`. Sa valeur est donc une référence vers un objet.

Les deux instructions suivantes ne font que rajouter une propriété (`hello`) à cet objet et en modifier une existante (`foo`) ; cependant l'objet est bien toujours le même.

Si plus tard le code tente de changer la **valeur** de `monObjet`, nous avons bel et bien une erreur :

```
(function() {
  const monObjet = { foo: 'bar' };

  monObjet.hello = 'world';
  monObjet.foo = 42;

  monObjet = { a: 1 };
  // => TypeError: Assignment to constant variable.

  // ou ...
  monObjet = 'boom';
  // => TypeError: Assignment to constant variable.
})();
```

Les objets littéraux

Il existe en Javascript plusieurs façon de créer un objet, la méthode standard étant d'utiliser la méthode `Object.create()`. Afin de simplifier la création d'objets simples, il existe également la syntaxe des objets littéraux.

Les objets littéraux sont ceux que l'ont crée le plus souvent en ES5 en utilisant la syntaxe :

```
{ {nom de propriété} : {valeur} [, {nom de propriété} : {valeur} ...] }
```

Par exemple :

```
const moObjet = {
  hello: 'world',
  solution: 42,
};
```

Ecmascript 6 accepte désormais de nouvelles syntaxes dans les objets littéraux.

Les méthodes

Les méthodes peuvent désormais s'écrire directement sans avoir à préciser `function` :

```
const monObjet = {
  maMethode1() {
    console.log('hello');
  },
  maMethode2(arg1, arg2) {
    console.log(arg2);
  }
};

monObjet.maMethode1();
// => "hello"
monObjet.maMethode2('banana', true);
// => true

// Equivalent ES5
const monObjetES5 = {
  maMethode1: function() {
    console.log('hello');
  },
  maMethode2: function(arg1, arg2) {
    console.log(arg2);
  }
};
```

Les valeurs implicites par homonymie

Lorsque l'on souhaite définir une propriété de l'objet dont le nom est égal à celui d'une variable dont on souhaite lui affecter la valeur, il suffit désormais d'écrire ce nom.

```
const count = 42;

const monObjet = {
  count
};

console.log(monObjet.count);
// => 42

// Equivalent ES5
const monObjetES5 = {
  count: count
};
```

Les noms de propriété dynamiques

Les noms de propriétés dans les objets litteraux peuvent désormais etre dynamiques en entourant une expression Javascript entre crochets (`[]`).

```
const count = 42;

function genererNom() {
  return 'Maurice';
}

const monObjet = {
  [genererNom().toUpperCase() + count] : 'le plombier'
};

console.log(monObjet.MAURICE42);
// => "le plombier"

// Equivalent ES5
const monObjetES5 = { };
monObjetES5[genererNom().toUpperCase() + count] = 'le plombier';
```

Les symboles

Les symboles sont un nouveau type de donnée primitif (comme `number`, `string` ou `object` ; comprendre *une des valeurs retournées par* `typeof`) introduit par Ecmascript 6.

Ils représentent une donnée qui est :

- **Unique** : Chaque symbole est unique de manière absolue, il n'aura jamais aucune autre variable qui lui soit égale

```
console.log(Symbol('name') === Symbol('name'));
// => false
```

- **Immuable** : La valeur d'un symbole est définie dès sa création et ne peut jamais changer

Les symboles sont créés en utilisant la syntaxe :

```
Symbol([description textuelle du symbole])
```

La description fournie en argument est optionnelle, elle ne sert qu'à aider à la compréhension du code ou à débugguer.

Ils sont utilisés comme identifiants de propriétés d'objets :

```
const monObjet = {
  [Symbol()] : 42,
  [Symbol('Mon super symbole')] : 42,
}
```

Pour ne pas exposer des propriétés

Prenons l'objet suivant :

```
const Chaton = (function() {
    function Chaton(nom) {
        this.nom = nom;
    }
    Chaton.prototype.miaule = function() {
        console.log('Le chat ' + this.nom + ' fait miaou');
    };
    return Chaton;
}());
```

Nous pouvons appeler la méthode `miaule()` :

```
const terton = new Chaton('terton');
terton.miaule();
// => Le chat terton fait miaou
// (humour douteux)
```

Cependant, l'objet expose sa propriété `nom` qui peut être modifiée depuis l'"éxterieur" :

```
const terton = new Chaton('terton');

terton.nom = 'lutier';
terton.miaule();
// => Le chat lutier fait miaou
// (humour toujours douteux)
```

Nous pouvons modifier l'objet de façon à utiliser un symbole (qui ne sera pas accessible depuis l'extérieur de la déclaration car déclaré avec `const`) pour stocker le nom :

```
const Chaton = (function() {
    const symbole = Symbol('nom');

    function Chaton(nom) {
        this[symbole] = nom;
    }
    Chaton.prototype.miaule = function() {
        console.log('Le chat ' + this[symbole] + ' fait miaou');
    };
    return Chaton;
}());
terton.miaule();
// => Le chat terton fait miaou
terton[Symbol('nom')] = 'lutier';
terton.miaule();
// => Le chat terton fait miaou
```

Tous les Symboles étant uniques, `terton[Symbol('nom')]` correspond donc à une toute nouvelle propriété.

Pour personnaliser le comportement des fonctions natives

Beaucoup d'aspects natifs du langage Javascript comme la résolution d'instanceof ou la vérification des expressions régulières sont désormais modifiables via des propriétés `Symbol`.

En effet, il aurait été dangereux auparavant de laisser accès à ces propriétés en les référençant par un simple nom ; par exemple, pour surcharger la méthode `match()` d'une chaîne, les risques d'écrasement avec un propriété nommée `match` auraient été trop grands.

L'objet `Symbol` possède donc des propriétés permettant d'accéder à des symboles connus.

Attention cependant, très peu de moteurs Javascript supporte ces Symbols connus pour l'instant, sauf Symbol.iterator (voir chapitre Iteration)...

Le registre global de symboles

L'objet symbole maintient un registre global (accessible depuis tous les contextes) de symboles utilisables via les méthodes :

`Symbol.for(nom)`

Retourne le symbole présent dans le registre sous ce nom ou, si il n'existe pas, en crée un et le retourne.

```
const symbol1 = Symbol.for('nom');
const symbol2 = Symbol.for('nom');
const symbol3 = Symbol.for('nom 2');

console.log(symbol1 === symbol2);
// => true
console.log(symbol1 === symbol3);
// => false
```

`Symbol.keyFor(symbole)`

Retourne le nom dans le registre du symbole passé en argument, si il est présent, sinon retourne `undefined`.

```
const symbol1 = Symbol.for('nom'); // Crée un symbole sous le nom "nom"
const symbol2 = Symbol('nom'); // A ne pas confondre avec la description

console.log(Symbol.keyFor(symbol1));
// => "nom"
console.log(Symbol.keyFor(symbol2));
// => undefined
```

Les collections et dictionnaires

En Javascript, collection est synonyme de `Array` et dictionnaire synonyme de `Object`, Ecmascript ajoute deux nouveaux types d'objet natifs pour en améliorer la gestion : Les objets `Set` et `Map`.

`Set`

Syntaxe :

```
new Set([ valeurs ]);
```

Le paramètre `valeurs` passé peut être tout objet itérable.

Un `Set` est un objet permettant de stocker un ensemble de valeurs, quelle qu'elle soit, comme un tableau `Array` ; cependant `Set` va les stocker de manière unique. Cela veut dire que si on essaye d'insérer plusieurs fois la même valeur dans un `Set`, celle-ci ne sera stockée qu'une seule fois.

Cette gestion native présente un réél intérêt par rapport à la gestion en `Array` ou l'on devait à chaque insertion vérifier l'existence de la valeur avec `indexOf()` et ou l'on devait utiliser `filter()` pour en supprimer. De plus, `Set` conserve les valeurs dans l'ordre ou elles ont été ajoutées lors d'itérations, ce qui n'était pas assuré avec `Array`.

Voici les propriétés et méthodes de `Set` :

- **`Set.prototype.size => number`**

  Permet de connaître la taille de l'ensemble, donc le nombre de valeurs y étant stockées.

- **`Set.prototype.add(valeur) => undefined`**

  Ajoute, si elle n'est pas déjà présente, une valeur dans l'ensemble.

- **`Set.prototype.has(valeur) => boolean`**

  Retourne `true` une valeur est présente dans l'ensemble, `false` sinon.

- **`Set.prototype.delete(valeur) => boolean`**

  Enlève, si elle est présente, une valeur de l'ensemble.

Retourne `true` si la valeur était présente, `false` sinon.

- **`Set.prototype.clear() => undefined`**

  Enlève toute les valeurs de l'ensemble.

- **`Set.prototype.forEach(fonctionIteration[, contexte]) => undefined`**

  Exécute la fonction `fonctionIteration` pour chacune des valeurs dans l'ensemble. La fonction sera appelée avec trois arguments :

  - La valeur de l'élément en cours d'itération
  - La valeur de l'élément en cours d'itération *(pour avoir la même signature que les objets `Map`)*
  - L'ensemble `Set` parcouru

  Si un argument `contexte` est passé, celui-ci sera assigné comme contexte lors de l'éxécution de la fonction *(valeur de `this`)*.

Comparaison d'unicité

Afin de tester si une valeur est déjà présente dans l'ensemble, un comparaison équivalente à `===` est effectuée sur chaque élément, avec une exception pour la valeur `NaN` *(pour `Set`, il est considéré que `NaN` **===** `NaN`)*

Exemple complet

```
const ensemble = new Set();

ensemble.add(1);
ensemble.add('youpi');
ensemble.add({});
ensemble.add(undefined);

console.log(ensemble);
// => Set { 1, 'youpi', {}, undefined }
console.log(ensemble.size);
// => 4

ensemble.add(NaN);
ensemble.add(NaN);
console.log(ensemble);
// => Set { 1, 'youpi', {}, undefined, NaN }

ensemble.add(0);
ensemble.add(+0);
ensemble.add(-0);
console.log(ensemble);
// => Set { 1, 'youpi', {}, undefined, NaN, 0 }
console.log(ensemble.size);
// => 6

console.log(ensemble.has(1));
// => true

ensemble.delete(5);
// => false
ensemble.delete(NaN);
// => true
console.log(ensemble);
// => Set { 1, 'youpi', {}, undefined, 0 }
console.log(ensemble.size);
// => 5

ensemble.forEach(console.log);
// => 1 1 Set { 1, 'youpi', {}, undefined, 0 }
// => ...
// => 0 0 Set { 1, 'youpi', {}, undefined, 0 }

ensemble.clear();
console.log(ensemble);
// => Set {}
console.log(ensemble.size);
// => 0
```

Map

Syntaxe :

```
new Map([ valeurs ]);
```

Le paramètre `valeurs` passé peut être tout objet itérable.

Un `Map` est un objet permettant de stocker des valeurs indexées par un clé unique, on peut le considérer comme un dictionnaire. Contrairement à un `Objet` où les noms de propriétés doivent être des chaînes, les clé et les valeurs d'un `Map` peuvent être de n'importe quel type.

Cette gestion native présente un réél intérêt par rapport à la gestion en `Object` dans le sens ou il n'y a pas de confusion ou de conflit possible avec les vraies propriétés d'un objet (prototype, etc.).

Voici les propriétés et méthodes de `Map` :

- **`Set.prototype.size => number`**

  Permet de connaître le nombre de clés présentes dans le dictionnaire.

- **`Set.prototype.set(clé, valeur) => undefined`**

  Ajoute, si la clé n'est pas déjà présente, une valeur dans le dictionnaire indexée par la clé. Si la clé est déjà présente, la valeur est remplacée.

- **`Set.prototype.has(clé) => boolean`**

  Retourne `true` la clé est présente dans le dictionnaire, `false` sinon.

- **`Set.prototype.delete(clé) => boolean`**

  Supprime, si la clé est présente dans le dictionnaire, la valeur indexée pour cette clé.

  Retourne `true` si la clé était présente, `false` sinon.

- **`Set.prototype.clear() => undefined`**

  Enlève toute valeurs du dictionnaire.

- **`Set.prototype.keys() => Iterator`**

  Retourne un objet itérateur des clés du dictionnaire.

- **`Set.prototype.forEach(fonctionIteration[, contexte]) => undefined`**

  Exécute la fonction `fonctionIteration` pour chacune des valeurs dans le dictionnaire. La fonction sera appelée avec trois arguments :

- La valeur de l'élément en cours d'itération
- La clé indexant la valeur
- Le dictionnaire `Map` parcouru

Si un argument `contexte` est passé, celui-ci sera assigné comme contexte lors de l'éxécution de la fonction *(valeur de `this`)*.

Comparaison d'unicité

Afin de tester si une clé est déjà présente dans le dictionnaire, une comparaison équivalente à `===` est effectuée sur chaque clé, avec une exception pour la valeur `NaN` *(pour `Map`, il est considéré que `NaN` **===** `NaN`)*

Exemple complet

```
const dictionnaire = new Map();

dictionnaire.set(1, 'youpi');
dictionnaire.set('youpi', 41);
dictionnaire.set('youpi', 42);
dictionnaire.set({}, new Set([1, 2]));

console.log(dictionnaire);
// => Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 } }
console.log(dictionnaire.size);
// => 3

dictionnaire.set(NaN, {});
dictionnaire.set(NaN, 101010);
console.log(dictionnaire);
// => Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 }, NaN => 101010 }

console.log(dictionnaire.has('youpi'));
// => true
console.log(dictionnaire.has({}));
// => false
// {} !== {}, ce sont deux instances d'objet différentes !


dictionnaire.delete(5);
// => false
dictionnaire.delete(NaN);
// => true
console.log(dictionnaire);
// => Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 } }
console.log(dictionnaire.size);
// => 3

dictionnaire.forEach(console.log);
// => 'youpi', 1, Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 } }
// => 42, 'youpi', Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 } }
// => Set { 1, 2 }, {}, Map { 1 => 'youpi', 'youpi' => 42, {} => Set { 1, 2 } }

dictionnaire.clear();
console.log(dictionnaire);
// => Map { }
console.log(dictionnaire.size);
// => 0
```

Les collections et dictionnaires volatiles

Les références et le ramasse-miettes

En Javascript, les variables dites "primitives" (donc qui ne sont pas des objets et qui n'ont pas de méthodes), à savoir les variables de type :

- `Number` (1, -42, 5.012, 2.6561398887587475e+95, NaN, Infinity, ...)
- `String` ('', 'okay', ...)
- `Boolean` (`true` et `false`)
- `Undefined`
- `Null`
- `Symbol`

sont des variables dites "simples", c'est à dire qu'elles contiennent la donnée elle même.

Au contraire, les variables de type `Object` dont des variables dites "références", c'est à dire qu'elles ne contiennent pas la donnée elle-même mais l'adresse en mémoire de cette donnée.

Ainsi, si l'exemple suivant :

```
let data = {};

data.prop = 42;
maFunction(data);
```

peut être traduit "vulgairement" en :

```
- Crée un objet et stocke le dans la mémoire.
- Crée une variable nommée data qui contient l'adresse en mémoire cet objet.
  => let data = {};

- Ajoute une propriété nommée prop dans l'objet référencé à cette adresse
- Assigne la valeur 42 à cette propriété
  => data.prop = 42;

- Exécute la fonction maFunction en lui passant en paramètre l'adresse réferencant l'obje
  => maFunction(data);
```

Ainsi, le code contenu dans `maFunction` récupèrera bien l'objet `data` et non une copie, c'est ce que l'on appelle le passage par référence.

Cette notion de référence est importante pour la gestion de la mémoire en Javascript, notamment lors du passage du ramasse-miettes).

Pour nettoyer l'espace occupé par les objets, celui-ci fonctionne en utilisant la méthode du comptage de références ; c'est à dire qu'à chaque passage, il compte le nombre de référence pointant vers l'adresse de chaque objet en mémoire, si le résultat est 0, il libére alors la place qu'occupait cet objet, on dit alors que l'objet est déréférencé.

Une des conséquences de ceci est que tant qu'un objet est référencé dans un `Map` ou un `Set`, il reste présent en mémoire et ne sera pas nettoyé par le ramasse-miette :

```
const monSet = new Set();

(function genere() {
  const referenceLocale = {};

  // Génération de données afin que l'objet utilise beaucoup de place en mémoire
  for (let i = 0; i < 100000; i++) {
    referenceLocale['prop_' + i] = 'donnée';
  }
  monSet.add(referenceLocale);
  console.log(typeof referenceLocale);
  // => Object
  // A la sortie de la fonction, referenceLocale n'existe plus mais l'objet
  // est toujours référencé dans monSet
})();

console.log(typeof referenceLocale);
// => Undefined
console.log(monSet.size);
// => 1
// La référence est toujours présente, les données restent en mémoire.
```

La solution des références faibles

ES6 introduit deux nouveaux types d'objet natifs : `WeakSet` et `WeakMap`. Ils sont l'équivalent de `Set` et `Map`, à la différence près qu'ils stockent des valeurs en utilisant des références faibles. Une référence faible a le meme comportement qu'une référence normale mais n'incrémente pas le compteur de référence vers l'objet qu'elle pointe.

Cela signifie que dès qu'il n'existera plus de référence "normale" vers un élément qu'ils contiennent (clé ou valeur), celui-ci sera supprimé par le ramasse-miettes.

Si l'on prend l'exemple précédent en remplace le `Set` par un `WeakSet` :

```
const monSet = new WeakSet();

(function genere() {
  const referenceLocale = {};

  // Génération de données afin que l'objet utilise beaucoup de place en mémoire
  for (let i = 0; i < 100000; i++) {
    referenceLocale['prop_' + i] = 'donnée';
  }
  monSet.add(referenceLocale);
  console.log(typeof referenceLocale);
  // => Object
  // A la sortie de la fonction, referenceLocale n'existe plus mais l'objet
  // est toujours référencé dans monSet
})();

console.log(typeof referenceLocale);
// => Undefined

// monSet est donc maintenant vide
```

Les `WeakSet` et `WeakMap` sont donc très utiles pour éviter les fuites mémoires puisqu'il n'est plus nécéssaire de d'utiliser l'instruction `delete` pour libérer la mémoire prise par un objet qui n'est plus utilisé.

En échange de cet avantage, ils présentent un inconvénient, il n'est pas possible de les énumérer ou de les compter ; le passage du ramasse-miette étant indeterminé au moment de l'exécution (il n'y a pas de facon *déterministe* de prédire les valeurs).

Les objets `WeakMap` et `WeakSet` n'ont donc pas de propriété `.size` ni de méthodes `.forEach()`, `.keys()`, `.values()`, etc.

Enfin, ils imposent d'utiliser des variables de type objet pour les clés des `WeakMap` et de ne stocker que des objets dans les `WeakSet`. Tout ajout d'un autre type de donnée (car n'étant pas des références) provoquera une erreur de type `TypeError`.

Exemple d'utilisation : Des propriétés "privées" d'un objet

Considérons le fichier suivant définissant un objet (en Node.js):

```
const donneesPrivees = new WeakMap();

function Personne() {
  const me = {
    age: 34,
    nom: 'John Doe',
  };
  privates.set(this, me);
}

Personne.prototype.getAge = function () {
  return(privates.get(this).age);
};
Personne.prototype.setAge = function (age) {
  privates.get(this).age = age;
};

module.exports = Personne;
```

L'utilisation de `WeakMap` permet de stocker les données propres à chaque objet grace à la ligne `privates.set(this, me);`. Dès lors qu'une instance de l'objet `Personne` sera déréférencée (donc plus utilisée), le ramasse-miette supprimera automatiquement les données stockées pour celui-ci dans `donneesPrivees`.

Itération

En développement, on appelle un processus itératif ; ou plus simplement, une itération ; une séquence d'instructions qui doit être exécutée pour chaque valeur d'un ensemble donné.

En Javascript, généralement, l'itération se fait sur un tableau ou un objet par l'intermédiaire de l'instruction for...in.

```
var tableau = ['vive le vent', 'vive le vent', "vive le vent d'hiver"];

for (var index in tableau) {
    console.log(tableau[index]);
}
```

Exécuter dans votre navigateur

Ces itérations, bien que très souvent utilisées, sont très limitées et ne permettent pas beaucoup de personnalisation.

Pour pallier à cette limitation, ES6 a complètement revu le processus d'itération afin d'offrir aux développeurs le contrôle de chaque étape et la manière dont les séquences sont exécutées et enchaînées.

Les objets `Iterator`

@todo définir d'abord let / const @todo définir d'abord Symbol @todo définir d'abord les objets litteraux

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...

To be done ...